МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»

Кафедра «Вычислительная математика и программирование»

**Лабораторная работа №2**

**по курсу «Параллельная обработка данных»**

**Обработка изображений на GPU. Фильтры.**

Выполнил: Лукашкин К.В.

Группа: 8О-408Б

Преподаватель: A.Ю. Морозов

Москва, 2019

**Условие**

Цель работы. Научиться использовать GPU для обработки изображений.

Использование текстурной памяти.

Вариант 3. Билинейная интерполяция.

**Программное и аппаратное обеспечение**

Характеристики графического процессора:

|  |  |
| --- | --- |
| Compute capability | 6.1 |
| Name | "GeForce GTX 1060 with Max-Q Design" |
| Total global memory | 6373572608 bytes |
| Shared memory per block: | 49152 bytes |
| Registers per block | 65536 |
| Max threads per block | 1024 |
| Max dimension size of a thread block (x,y,z) | (1024, 1024, 64) |
| Max dimension size of a grid size (x,y,z) | (2147483647, 65535, 65535) |
| Total constant memory | 65536 bytes |
| Multiprocessors count | 10 |

Процессор: Intel Core i5-8300H @ 8x 4GHz

|  |  |
| --- | --- |
| Number of cores | 4 |
| Number of threads | 8 |
| Processor Base Frequency | 2.30 GHz |
| Max Turbo Frequency | 4.00 GHz |
| Cache | 8 MB SmartCache |
| Bus Speed | 8 GT/s DMI |

Оперативная память: 8Gb DDR4

Частота: 2666 МГц

Жёсткий диск: 128Gb SSD

ATA SanDisk X600 M.2

OS: Ubuntu 18.04 bionic

Kernel: x86\_64 Linux 5.0.0-29-generic

Compiler:

nvcc: NVIDIA (R) Cuda compiler driver

Cuda compilation tools, release 9.1, V9.1.85

IDE: Текстовый редактор Atom

Версии компонентов:

Atom : 1.40.1

Electron: 3.1.10

Chrome: 66.0.3359.181

Node : 10.2.0

**Метод решения**

Необходимо реализовать изменение размера изображения, используя

билинейную интерполяцию. Значения пикселей привязываются к их центрам, а не к

углам.

Для хранения изображения я использую встроенную структуру *texture.* С параметром *addressMode* равной *cudaAddressModeClamp* по обоим осям изображения. Это позволяет получать значения пикселей по координатам со значениями меньше 0 или больше границы изображения, в таком случае они будут равны значению пикселя на крае изображения.

Поскольку использование встроенной текстурной билинейной интерполяции запрещено, в обязательном порядке у текстуры ставится *filterMode* со значением *cudaFilterModePoint*, при обращении к нецелым координатам не будет производится интерполяция, а будет выдано значение ближайшего пикселя. Однако в ходе выполнения программы обращение происходит только по целым значениям координат, то гарантируется округлением координат с помощью функции floorf.

При билинейной интерполяции для вычисления цветов дополнительных пикселей относительно основных, исходных в оригинальном изображении с известными цветовыми координатами. Конкретно моя реализация использует 4 ближайших пикселя в оригинальном изображении, которые образуют квадрат заключающий в себя вычисляемый пиксель. При этом отсчёт идёт от середины пикселя.

**Описание программы**

* \_\_global\_\_ void kernel– ядро, выполняет масштабирование изображения с помощью билинейной интерполяции.
* CSC(call) – макрос, выполняет обработку возникающих ошибок в ходе выполнения функций CUDA, выводит справочную информацию об ошибке.
* read\_image – считать изображение во внутреннюю память
* write\_image – вывести полученное изображение во внешний файл
* initialize\_texture – инициализировать объект texture, он связывается с массивом cuda исходного изображения с помощью cudaBindTextureToArray.

**Результаты**

Запуск непосредственно алгоритма в программе просходит в виде: kernel<<<numBlocks, threadsPerBlock>>>

Небольшой тест (Изображение 100х100)

Будем сравнивать увеличение и уменьшение изображения в 2 раза

|  |  |  |  |
| --- | --- | --- | --- |
| **Количество блоков** | **Количество нитей на блок** | **Время x2 , мс** | **Время x0.5 , мс** |
| 1 | 32 | 0.614080 | 0.068192 |
| 128 | 128 | 0.083520 | 0.404736 |
| 256 | 256 | 0.161504 | 0.024768 |
| 256 | 512 | 0.040256 | 0.028960 |
| 512 | 1024 | 0.048416 | 0.344128 |

Cредний тест (Изображение 1000х200)

|  |  |  |  |
| --- | --- | --- | --- |
| **Количество блоков** | **Количество нитей на блок** | **Время x2 , мс** | **Время x0.5 , мс** |
| 1 | 32 | 15.955840 | 0.907072 |
| 128 | 128 | 0.243872 | 0.038464 |
| 256 | 256 | 0.243488 | 0.042880 |
| 256 | 512 | 0.244992 | 0.042912 |
| 512 | 1024 | 0.280256 | 0.053664 |

Предельный тест

Будем увеличивать изображение до 20000x20000.

В таблице указано исходное изображение.

|  |  |  |  |
| --- | --- | --- | --- |
| **Количество блоков** | **Количество нитей на блок** | **Время, мс**  **104х104** | **Время, мс 103x103** |
| 1 | 32 | 5554.287109 | 4786.014160 |
| 128 | 128 | 102.670403 | 110.045792 |
| 256 | 256 | 101.103584 | 109.977661 |
| 256 | 512 | 79.855774 | 91.396835 |
| 512 | 512 | 94.354019 | 108.954659 |
| 1024 | 1024 | 101.114944 | 106.653214 |

Для самого малого теста самой быстрой оказалась конфигурация 1,32 с одним блоком и 32 нитями, с увеличением блоков, скорость выполнения только увеличивается.

Для средних же тестов 128,128 и 256,256 оказались быстры. При этом при уменьшении изображения 128,128 показала себя значительно лучше.

Для больших тестов алгоритм при конфигурации 256, 512 показал лучший результат который с увеличением блоков и нитей не улучшался, но и не ухудшался.

Результаты показали что задача в данном виде является I/O bound, поскольку реальное время выполнения программы в несколько раз превышает время работы непосредственно алгоритма, т. е. большая часть времени тратиться на ввод-вывод.

Хоть время на ввод/вывод не включалось в таблицы, при выполнении тестов время потраченное на запись результата было вполне ощутимо и без точных замеров.

**Сравнение с CPU.**

Программа с CUDA, запускается на оптимальной конфигурации, полученной в тестах ранее. Время в миллисекундах.

|  |  |  |
| --- | --- | --- |
| **Размер теста** | **CPU** | **GPU** |
| Небольшой | 1.839000 | 0.083520 |
| Средний | 36.066000 | 0.007936 |
| Предельный | 18044.394000 | 3.459488 |

Исходя из результатов теста, алгоритм на GPU работает значительно быстрее.

**Пример работы с изображениями**

Увеличение с изменениями пропорций



****

Уменьшение изображения



****

**Выводы**

Проделав лабораторную работу, я изучил и применил алгоритм билинейной интерполяции, реализовал его параллельную версию на GPU. Обработка изображений одна из основных задач для которых GPU было создано, высокая степень распараллеливания задачи, позволяет достигать высокой скорости работы алгоритма. Билинейная интерполяция в большинстве случаев принименяется именно при решении задач ресемплинга – масштабирования.